A Massively Parallel Associative Memory Based on Sparse Neural Networks

Zhe Yao[*], Vincent Gripon[†] and Michael G. Rabbat[*]

Abstract—Associative memories store content in such a way that the content can be later retrieved by presenting the memory with a small portion of the content, rather than presenting the memory with an address as in more traditional memories. Associative memories are used as building blocks for algorithms within database engines, anomaly detection systems, compression algorithms, and face recognition systems. The classical example of an associative memory is the Hopfield neural network. Recently, Gripon and Berrou have introduced an alternative construction which builds on ideas from the theory of error correcting codes and which greatly outperforms the Hopfield network in capacity, diversity, and efficiency. In this paper we implement a variation of the Gripon-Berrou associative memory on a general purpose graphical processing unit (GPU). The work of Gripon and Berrou proposes two retrieval rules, SUM-OF-SUM and SUM-OF-MAX. The SUM-OF-SUM rule uses only matrix-vector multiplication and is easily implemented on the GPU. The SUM-OF-MAX rule is much less straightforward to implement because it involves non-linear operations. However, the SUM-OF-MAX rule gives significantly better retrieval error rates. We propose a hybrid rule tailored for implementation on a GPU which achieves a 760-fold speedup without sacrificing any accuracy.

Index Terms—Associative memory, Recurrent neural networks, Parallel processing, High performance computing, Sparse coding, CUDA, GPGPU

I. INTRODUCTION

We are all familiar with conventional memory systems where the address space and the information content stored in the memory are kept separate. For instance, given a mailbox number, we can fetch the parcels inside, and in a modern computer, the CPU retrieves a stored integer from RAM by accessing a specified 32- or 64-bit hardware address.

An associative memory is a device or data structure that maps input patterns to output patterns. It differs from conventional memory systems in that there are no explicit addresses. Associative memories store patterns. Then, given an input pattern, the associative memory produces the paired output pattern. Since no explicit address is involved in its operation, the content of the input pattern itself associates directly with the paired output pattern, from which the name associative memory originates. Although associative memories could be implemented using conventional memory systems, neural networks, dating back to Hopfield networks, have been used as associative memories which retrieve patterns without having to search through the stored pattern space. It is worth noting that

* Z. Yao and M.G. Rabbat are with the Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada. Email: zhe.yao@mail.mcgill.ca, michael.rabbat@mcgill.ca.
† V. Gripon is with the Electronics Department, Télécom Bretagne, Brest, France. Email: vincent.gripon@telecom-bretagne.eu.

hash tables, implemented using conventional memory systems, resemble associative memories since they map keys (inputs) to values (outputs), but still an explicit address needs to be generated first.

Associative memories can be categorized into two different types [1]: hetero-associative and auto-associative. In hetero-associative memories, the input and output patterns can be distinct in their nature or format (e.g., Rosenblatt's perceptron [2], or Kohonen's self-organizing map [3]). In auto-associative memories, the input and output patterns have the same format. This paper focuses on auto-associative memories.

Associative memories have applications in a variety of domains. For instance, in communication networks, routers need to quickly determine which port an incoming frame should be forwarded to based on the destination IP address. In signal and image processing, one commonly needs to match noisy or corrupted data to a predefined template. Similar tasks appear in database engines [4], anomaly detection systems [5], data compression algorithms [6], face recognition systems [7] and many other machine learning frameworks.

A. Historical Background and Related Work

Associative memories have a long history within the field of neural networks. Associative memories provide two operations: storing and retrieving. In the storing operation, pairs of patterns are fed into the memory and the internal connections between neurons are modified, forming an aggregated representation of the pairs that have been stored. In the retrieving operation (also referred to as "decoding"), the associative memory is presented with a probe pattern (input), which may be a corrupted or modified version of the stored pattern, and the memory should retrieve the most relevant or related pattern that was previously stored in a quick and reliable manner.

The linear associator [8] is one of the simplest and earliest associative memory models; see Fig. 1 for an illustration. A linear associator has two layers of neurons—the input layer and the output layer. Synapses only exist between these two layers, hence the network can be viewed as a bipartite graph. Edges or connections in the network are directed from input to output neurons. The number of neurons in each layer can be different in general, so the linear associator can be used as both an auto-associative and a hetero-associative memory. While storing patterns, the linear associator modifies link weights according to the Hebb's rule [9]. Then, while decoding a pattern, the network is presented with a given input pattern, and the paired output pattern is retrieved from the output layer immediately after one step of feed forward computation. Since

the paired pattern depends on a linear combination of the input pattern values, if all the input patterns are pairwise orthogonal, then the linear associator can reconstruct the paired patterns perfectly. However, in most cases the orthogonality does not hold, thus the network diversity (i.e., the number of paired patterns that the network can store) is extremely low.
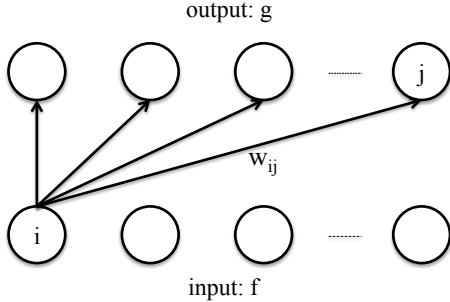

Fig. 1. An example of a linear associator network. Only the synapses of the first neuron in the input layer are drawn. A weight w_{ij} is assigned to the synapse between neuron i and neuron j.

John J. Hopfield's seminal work [10], [11] on associative memories brought these structures to the attention of the neural network community in the early 1980's. Fig. 2 shows an example of a Hopfield network, which is a bidirectional complete graph. Instead of being separated into different layers, Hopfield networks are comprised of only one layer which acts as both input and output. Retrieval of a pattern from the network works recurrently; i.e., when an impulse enters the network, the (output) values at iteration t serve as the input values at iteration $t+1$, and the values iterate until the network reaches its stable configuration if it ever converges. Since the layers are not differentiated, the Hopfield network can be only used as an auto-associative memory.
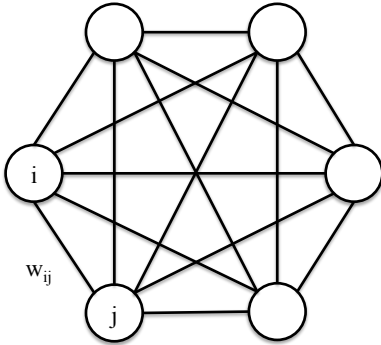

Fig. 2. An example of a Hopfield network with 6 neurons. w_{ij} is the weight associated with the synapse between neuron i and neuron j.

Kosko [12] extends the Hopfield network into a two-layer bidirectional associative memory (BAM). BAMs are different from linear associators since the edges in a BAM are not directed, and the retrieval rule is different. In a BAM, values at the output and input iterate until an equilibrium is reached. The BAM is also different from the Hopfield network since it incorporates distinct input and output layers. As a result, the BAM can be used as both a hetero-associative and an auto-associative memory, filling the gap between the linear associator and Hopfield networks.

The recent work of Gripon and Berrou proposes a new family of sparse neural network architectures for associative memories [13], [14]. We refer to these as *Gripon-Berrou neural networks* (GBNNs). The GBNN combines the notion of recurrence from Hopfield networks with ideas from the field of error correcting codes, and GBNNs achieve nearly optimal retrieval performance [13], [14]. A detailed description of the GBNN architecture and operation is given in Section II.

The GBNN is not the first attempt to link the associative memory with error correcting codes. For example, Berrou and Gripon [15] successfully introduce a set of Walsh-Hadamard codes in the framework of BAMs. The same authors also consider the use of sparse coding in a Hopfield network. They show that, given the same amount of storage, the GBNN outperforms conventional Hopfield networks in diversity, capacity (i.e., the maximum amount of stored information in bits) and efficiency (i.e., the ratio between capacity and the amount of information in bits consumed by the network when density reaches its maximum), while decreasing the retrieval error. In [16], they interpret GBNN using the formalism of error correcting codes, and introduce a new retrieval rule which further decreases the error rate. Jiang et al. [17] modify the GBNN structure to learn long sequences by incorporating directed edges into the network. Aliabadi et al. [18] make the extension to learn sparse messages.

The literature mentioned in the paragraphs above focuses on studying theoretical properties of GBNNs. To be useful in many applications, it is also essential to develop fast and efficient implementations of GBNNs. In [19], Jarollahi et al. demonstrate a proof-of-concept implementation using an FPGA. Due to hardware limitations, the implementation in [19] is constrained to have at most 400 neurons. This paper also focuses on the efficient implementation of GBNNs as associative memories, which supports a much larger number of neurons. All the existing algorithms can hopefully benefit from the exciting result we present.

B. Contributions

The primary contribution of this paper is to demonstrate an implementation of GBNNs on a GPU using CUDA. Our massively parallel implementation is $760\times$ faster, without any loss of retrieval accuracy, than a CPU implementation using optimized C++ libraries for linear algebra operations.

Towards developing an efficient parallel GBNN implementation, we study retrieval rules: SUM-OF-SUM and SUM-OF-MAX, which have been previously proposed in [14] and [16]. The SUM-OF-SUM rule is fast to implement because it requires only matrix-vector multiplications. The SUM-OF-MAX rule is slower because it involves non-linear operations, but it gives superior retrieval performance (lower error rates). We illustrate that, although it is faster, the SUM-OF-SUM rule can lead to oscillations, which is problematic. We also prove that the SUM-OF-MAX rule is guaranteed to converge, and we derive properties of both rules.

The tremendous speedup mentioned above comes from two main sources. First, we exploit the highly parallel architecture of the GPU to carry out operations efficiently.

Second, we develop a hybrid retrieval scheme using aspects of both SUM-OF-SUM and SUM-OF-MAX, which is tailored to parallel decoding architectures. Although we focus on a GPU implementation, we expect that the ideas presented here can be used to accelerate associative memory implementations on other parallel architectures.

C. Paper Organization

The rest of this paper is structured as follows. Section II reviews the GBNN associative memory architecture. Section III reviews the SUM-OF-SUM and SUM-OF-MAX retrieval rules. Section IV presents the proposed acceleration techniques and discusses the customized CUDA kernel functions which implement these techniques. Section V provides theoretical analysis and discussion of some properties of the retrieval rules considered in this work. Section VI proposes the novel hybrid retrieval rule. Section VII presents experimental results demonstrating the significant performance improvements obtained using GPUs. The paper concludes in Section VIII.

II. GRIPON-BERROU NEURAL NETWORKS (GBNNs)

In a GBNN, a pattern (a.k.a. "message") is divided into a tuple of smaller sub-messages. Specifically, we divide the pattern or message M into C sub-messages, $M = (m_1, m_2, \ldots, m_C)$, where each sub-message m_c takes values in a finite set of size L. For example, English words of length 10 characters could be represented as 10 sub-messages from an alphabet of size 26; alternatively, they could be represented as 5 sub-messages from an alphabet of size 26^2. Similarly, in an image, a sub-message could correspond to the intensity of a specific pixel, or to the collective intensities of a patch of pixels. Here we work in the abstract setting of messages and sub-message defined above; precisely how the associative memory is used is application-dependent.

For messages with C sub-messages, each taking one of L values, as described above, a GBNN [14] architecture consists of $n = CL$ binary-valued (0 or 1) neurons. The neurons are grouped into C clusters of L neurons each, and edges only exist between neurons in different clusters. A message $M = (m_1, \ldots, m_C)$ is represented in the network by activating (i.e., setting to 1) one neuron in each cluster corresponding to the value of m_c, and setting all other neurons to 0. In this way, the message is naturally encoded as a binary string of length L.

When a network is initialized, all edge weights are set to zero (equivalently, there are no edges in the network). When storing a message, we add edges to the network connecting all pairs of nodes which are activated for the particular message. For example, consider the network depicted in Fig. 3, where each message contains $C = 4$ sub-messages and each sub-message takes one of $L = 16$ different values. Let us use the convention that clusters are numbered from left to right and from top to bottom, so that ◯'s are cluster 1, □'s are cluster 2, and so on; let us use the same convention within each cluster so that the neurons within each cluster are numbered from $1, 2, 3, 4$ in the first row, and so on. The message indicated by the bold edges is $(9, 4, 3, 10)$. The edges corresponding to

any single message stored in the network thus correspond to a *clique*, since the neurons connected for that message form a complete sub-graph. The binary code that represents the bold clique in Fig. 3 reads 0000000010000000 0001000000000000 0010000000000000 0000000001000000.
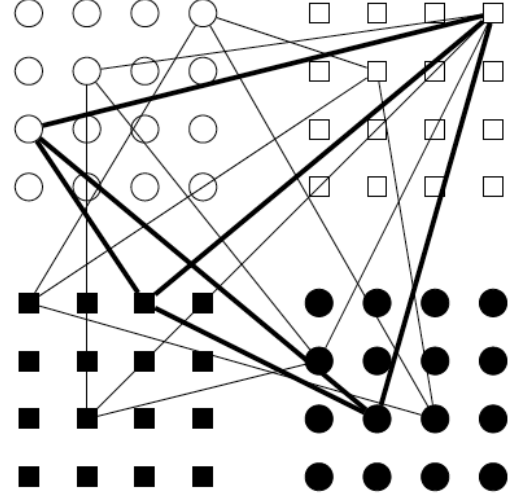

Fig. 3. An example of a network with 4 clusters of 16 neurons each. We number the clusters from left to right and from top to bottom as $1 \cdots 4$. The same scheme applies for neurons $1 \cdots 16$ within each cluster.

One key difference between GBNNs and other associative memory models is that edge weights only take the values 0 and 1; i.e., either an edge is present, or it is not. If a new message is being stored which involves an edge already existing in the network, the state of that edge is not changed (i.e., the edge remains present, and no new weight is assigned). In contrast, the weights of edges in, e.g., a Hopfield network depend on the number of messages which have been stored involving the corresponding pair of neurons. Hence, a GBNN initially contains no edges, and as more messages are stored, more edges are added to the network.

For retrieval, the network is presented with a partially erased message as a probe, e.g., $(m_1, m_2, ?, ?)$, and it must determine which (if any) stored message matches this input best. In this paper we focus on the case where only partial messages are presented for retrieval. If the network is presented with an entire message as a probe, then the problem boils down to deciding whether or not this message has been stored. For this case, it has been shown [14] that the missed detection rate is zero (i.e., messages which were previously stored are always recognized by the GBNN), and the false positive rate depends on the number of messages which have been stored in the network; see [14] for more details.

When a GBNN is probed with a partially erased message, such as $(m_1, m_2, ?, ?)$, initially neurons m_1 in cluster 1 and m_2 in cluster 2 are activated, and an iterative decoding procedure is used to determine the values of the other clusters. The next section discusses two possible retrieval rules.

III. RETRIEVAL RULES

In this section, we review two existing retrieval rules for GBNN, i.e., SUM-OF-SUM and SUM-OF-MAX.

A. *The* SUM-OF-SUM *Rule*

The simplest retrieval rule is to add all the signals a neuron receives in the current iteration. At the retrieval stage, when presented with a partially erased message, we initialize the network by deactivating (i.e., setting to 0) all the neurons within the clusters associated with erased sub-messages. We then repeat the following iterations. First, each neuron compute the sum of all connected neurons which are presently active. Then the neurons within each cluster with the most active connected neurons remain activated at the beginning of the next iteration. This "winner-takes-all" operation is the default retrieval rule proposed in [14].

More formally, let $\text{neuron}(c, l)$ denote the lth neuron in the cth cluster, and let $w_{(cl)(c'l')}$ denote an indicator variable for whether or not a connection is present between $\text{neuron}(c, l)$ and $\text{neuron}(c', l')$; i.e.,

$$w_{(cl)(c'l')} = \begin{cases} 1 & \text{neuron}(c, l) \text{ is connected to neuron}(c', l') \\ 0 & \text{otherwise} \end{cases} . \tag{1}$$

We also denote by s_{cl}^t and v_{cl}^t respectively the score function for the number of signals $\text{neuron}(c, l)$ receives and the indicator function for whether or not $\text{neuron}(c, l)$ is activated at iteration t, with v_{cl}^0 being the corresponding value for $\text{neuron}(c, l)$ in the probe; i.e.,

$$v_{cl}^t = \begin{cases} 1 & \text{neuron}(c, l) \text{ is activated in iteration } t \\ 0 & \text{otherwise} \end{cases} . \tag{2}$$

As a consequence, the retrieval procedure can be formalized as

$$s_{cl}^t = \gamma v_{cl}^t + \sum_{c'=1}^{C} \sum_{l'=1}^{L} \left(v_{c'l'}^t w_{(c'l')(cl)} \right) \tag{3}$$

$$s_{c,\max}^t = \max_{1 \le l \le L} s_{cl}^t \tag{4}$$

$$v_{cl}^{t+1} = \begin{cases} 1 & \text{if } s_{cl}^t = s_{c,\max}^t \\ 0 & \text{otherwise} \end{cases}, \tag{5}$$

where $\gamma \ge 0$ is reinforcement factor. Essentially, Eq. (3) counts the score for each neuron. It involves summing over all clusters and all neurons within each cluster, hence the name SUM-OF-SUM. Eq. (4) finds the value of the neurons with the strongest signal in each cluster, and Eq. (5) keeps them activated.

At the retrieval stage, the variables $w_{(cl)(c'l')}$ are fixed. These binary-valued variables are only changed when storing new messages. The only parameter to be tuned for retrieval using the SUM-OF-SUM rule is γ, which influences the extent to which a neuron's own value influences its signal at the current iteration.

B. *Problems with the* SUM-OF-SUM *Rule*

The SUM-OF-SUM rule, although straightforward and natural, might lead to unnecessary errors. This is due to the fact that during iterations, after evaluating Eq. (5), there might be multiple neurons in one cluster which all achieve the maximum value $s_{c,max}^t$ simultaneously. In this case, all these neurons will
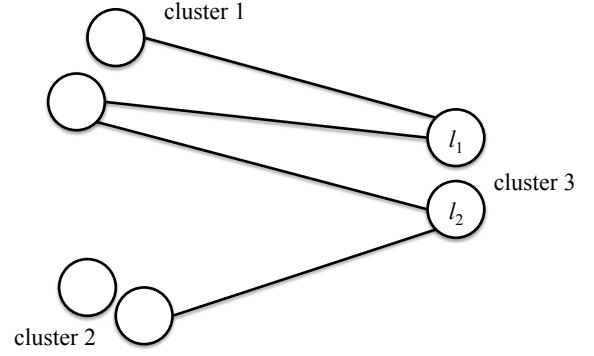

Fig. 4. Illustration of the SUM-OF-SUM trap. Only the signals flowing into cluster 3 are drawn.

stay activated and contribute to the signal strengths in the next iteration.

Consider the scenario shown in Fig. 4, where two neurons l_1 and l_2 both receive the same number of signals. Neuron l_1 receives two signals from cluster 1, while l_2 receives one signal from each cluster. In this case, l_2 should be favored, because we know that for any individual pattern that has been stored, only one neuron in each cluster should be activated. A possible but worse situation arises when l_1 receives more signals than l_2, since then l_1 will be the only activated neuron in this cluster at the beginning of the next iteration, even if l_2 was actually the correct neuron in cluster 3. An increasing number of clusters will complicate the problem even further. This can also cause the SUM-OF-SUM rule to diverge as will be shown in later sections.

C. *The* SUM-OF-MAX *Rule*

To avoid the problem mentioned in the previous subsection, the SUM-OF-MAX rule is proposed in [16]. The rule is formally described as follows:

$$s_{cl}^t = \gamma v_{cl}^t + \sum_{c'=1}^{C} \max_{1 \le l' \le L} \left(v_{c'l'}^t w_{(c'l')(cl)} \right) \tag{6}$$

$$v_{cl}^{t+1} = \begin{cases} 1 & \text{if } s_{cl}^t = \gamma + C - 1 \\ 0 & \text{otherwise} \end{cases} . \tag{7}$$

Eq. (6) involves a summation over *max* operation, hence the name SUM-OF-MAX. The basic idea is that, in order to retrieve the correct message, the score of a neuron should not be larger if it receives two or more signals from the same cluster, and the maximum taken in Eq. (6) ensures each neuron receives at most one signal from each cluster. Since each stored message corresponds to a clique of C neurons, one in each cluster, a neuron should be activated if it receives *exactly* $C - 1$ signals from the other clusters plus some value γ from the self loop.

In order for SUM-OF-MAX to work properly, the network must be initialized appropriately when a probe is presented for retrieval. Instead of initializing all neurons associated with erased sub-messages to be 0 as in SUM-OF-SUM, we initialize them to be 1. In that case, other neurons will definitely receive signals from these missing clusters, L signals per missing cluster, but they will be regulated by Eq. (7).

IV. Accelerating Retrieval

In this section, we will first briefly introduce the CUDA architecture. We discuss different approaches to speeding up the GBNN retrieval procedure in general, and then we focus on specific techniques for SUM-OF-SUM and SUM-OF-MAX separately. We also illustrate graphically the dedicated CUDA kernel functions for both rules.

A. CUDA

The *Compute Unified Device Architecture* (CUDA), introduced in 2007, is NVIDIA's computing platform solution to *general purpose computing on graphics processing units* (GPGPU), which enables dramatic increases in computing performance by harnessing the massively parallel resources of GPUs.

The basic programming pattern in CUDA is as shown in Fig. 5, where CPUs play the role of managers, invoking on the GPUs some computational intensive functions called *kernel* functions. After the kernel function is executed on the GPU, the CPU collects the results back to the host and then may invoke more kernel functions if necessary. Although a GPU can spawn many threads working simultaneously, each thread must run the same sequence of instructions. Kernel functions, and hence GPU computing in general, fit the category of "single instruction multiple data" (SIMD) [20] parallel computing platforms. The data are transferred back and forth between the CPU and GPU over the (slow) PCI or PCIe bus, one of the performance bottlenecks. Unfortunately, since the code control flow is on the CPU side, the time-costly transfers between the host and the video card are inevitable. Therefore keeping the transfer of data to a minimum is one of the crucial concerns.
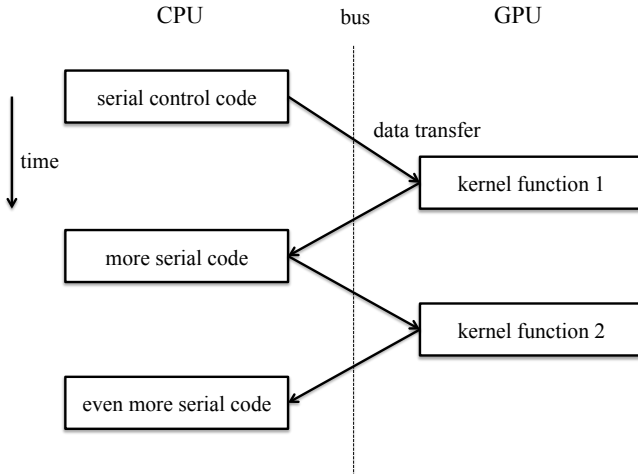

Fig. 5. CUDA programming scheme.

B. General Tricks

1) Vectorization: Although GBNN is a recurrent model, conceptually we can treat it as a layered network nevertheless. We repeat each iteration t as one layer, so that the number of layers can grow as large as the network needs to converge.

Let T denote the the total number of iterations to be run. The only two constraints to be satisfied are

$$w_{ij}^1 = w_{ij}^2 = \cdots = w_{ij}^T \quad \text{and} \quad w_{ij}^t = w_{ji}^t, t = 1, 2, \cdots, T.$$

The benefit is to borrow the matrix notation from layered networks which is more efficient in the parallel implementation. We map the original clustered structure into a flat space, where $\text{neuron}(c, l)$ becomes $\text{neuron}(i)$, with $i = (c-1)L+l$, ranging from 1 to $n = CL$. Then Eq. (1) and Eq. (2) can be rewritten as

$$w_{ij} = \begin{cases} 1 & \text{neuron}(i) \text{ connects neuron}(j) \\ 0 & \text{otherwise} \end{cases} \quad (8)$$

$$v_i^t = \begin{cases} 1 & \text{neuron}(i) \text{ is activated in iteration } t \\ 0 & \text{otherwise} \end{cases}. \quad (9)$$

We consider the edge weights w_{ij} as elements of an $n \times n$ matrix W, and neuron potentials v_i as elements of a vector $v \in \{0, 1\}^n$. Taking into account the reinforcement factor γ, we can rewrite Eq. (3) as

$$s^t = Wv^t, \quad (10)$$

with W being a symmetric matrix whose diagonal elements are all equal to γ and whose off-diagonal elements are all binary valued; i.e.,

$$W = \begin{pmatrix} \gamma & w_{12} & \cdots & w_{1n} \\ w_{21} & \gamma & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & \gamma \end{pmatrix}.$$

Thus, the score equation (10) is a matrix-vector product which can be computed efficiently in CUDA.

2) Batch Retrieval: A straightforward extension to vectorization is to bundle and process K probes simultaneously. To do so, we collect the K test messages into a value matrix

$$V^t = \begin{pmatrix} v^t(1) & v^t(2) & \cdots & v^t(K) \end{pmatrix},$$

with each column $v^t(k)$ being a value vector in Eq. (10), so that Eq. (10) becomes

$$S^t = WV^t. \quad (11)$$

Instead of retrieving messages sequentially, one after another, we aggregate K messages together and feed them into the GPU card at one shot, which will again make performance improvements. In particular, speedups are achieved using this approach because it allows us to exploit the SIMD nature of GPUs. It can also be more efficient to perform one large I/O transfer over the bus rather than multiple smaller transfers.

Batch retrieval arises naturally in applications where simultaneous retrievals are preferred. For instance, in face recognition, an associative memory can be used to recognize face features even areas are obstructed by sun glasses or a scarf. If we treat each image as a single message, the hardware requirement is simply prohibitive. A 256-level gray image of the size 512×512 requires an adjacency matrix W of $(2^9 \times 2^9 \times 2^8)^2 = 2^{52}$ elements. Alternatively, we can divide the image into smaller patches, treat each patch as a different

message, and process them in parallel. For another example, consider a network anomaly detection algorithm where we are given a batch of IP addresses, and we would like to check whether each belongs to a predefined blacklist. In Section VII below, we will refer to Eq. (11) as parallel decoding and Eq. (10) as serial decoding.

3) Reduction: Reduction refers to an operation that aggregates a vector of elements into a scalar (e.g., *sum*, *max* and *min*). In SUM-OF-SUM, the *max* operation is needed when evaluating Eq. (4) to determine which neurons remain active in the next iteration. In both rules, when deciding whether or not the retrieval procedure has converged, we need to compare two long vectors v^t and v^{t+1} of length n, and test if all of the neuron values stay unchanged. This reduction operation can be done in an efficient manner as illustrated in Fig. 6, where we invoke n threads in the first step, afterwards halving the number of threads in every successive operation. The time complexity thus decreases from $O(n)$ to $O(\log_2 n)$.
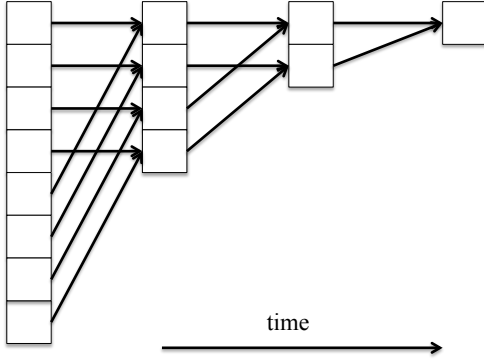

Fig. 6.   Parallel reduction operation scheme.

4) Sparsification: Memory access is an expensive operation on the video card, where both reading from and writing to memory cells are much slower than on the host CPU. In order to combat this inefficiency, we can reduce the number of memory accesses by accounting for the fact that GBNN is actually a sparse network; i.e., for a given message, ideally only one neuron should be activated for each cluster. Typically, the network structure should also be sparse, so we could implement both W and V^t as sparse matrices using compressed format, where we only record nonzero elements and their coordinates. Then evaluating Eq. (3) and Eq. (6) requires much less terms. However, the compressed format does not lead to a significant performance gain for both rules — SUM-OF-MAX benefits from the sparsification, while SUM-OF-SUM does not. The reason is that the dense matrix product in Eq. (11) for SUM-OF-SUM is an optimized operation on the GPU, whereas the compressed format deviates from the optimized pattern. Moreover, since V^t changes from one iteration to the next, it is not economical to implement V^t using compressed format either. On the contrary, W is fixed at the retrieval stage. We use a sparse matrix representation W only for the SUM-OF-MAX rule. Detailed numerical results are presented in Section VII.

C. Accelerating the SUM-OF-SUM Rule

The pseudocode for the SUM-OF-SUM procedure is given in Algorithm 1. It requires as inputs the maximum number of iterations permitted t_{max}, the weight matrix W with all of the clique structures preserved during the storing stage, and the message matrix V^0, with the kth column being the value vector for test message k and the erased clusters deactivated. On Line 4, S^t is the score matrix for iteration t, where the kth column is the score vector of length n for test message k. On Line 5, the kernel function takes S^t as input and essentially produces V^{t+1} by evaluating Eq. (4).

Algorithm 1 The SUM-OF-SUM retrieval procedure.

Input: the maximum number of iterations permitted t_{max}, the weight matrix W, the message matrix V^0 with each column as a partially erased message for recovery

Output: the recovered matrix V^t

1: t = -1
2: **repeat**
3:    t = t+1
4:    $S^t = W V^t$
5:    V^{t+1} = the kernel function as in Fig. 7
6: **until** $V^{t+1} == V^t$ or $t == t_{max}$

The first two columns of S^t are drawn in Fig. 7. In this particular example, each message can be divided into $C = 3$ clusters. In our implementation, a dedicated thread processes one cluster, finding the maximum value within that cluster, and then keeps the neurons that reach the maximum value activated. Assuming that there are K messages to be recovered, a total of CK threads are used. The retrieval procedure terminates when either the network converges or it reaches the maximum number of iterations permitted.
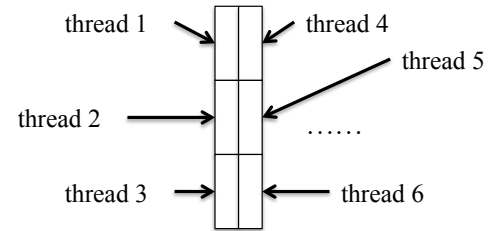

Fig. 7.   Illustration of the kernel function for the SUM-OF-SUM rule. The first two columns of S^t are drawn. Each rectangular represents a cluster with L neurons. A thread will determine the maximum value in its cluster and set the corresponding neurons activated.

D. Accelerating the SUM-OF-MAX Rule

The pseudocode for SUM-OF-MAX is almost the same with Algorithm 1, except that Lines 4 and 5 are replaced by another kernel function illustrated in Fig. 8. In order to better explain the concept, the serial decoding of a single message v^t is presented here, where the same number of threads are needed as the number of neurons n in the network. The extension to the parallel decoding scheme of K bundled messages is straightforward, where nK threads are needed.
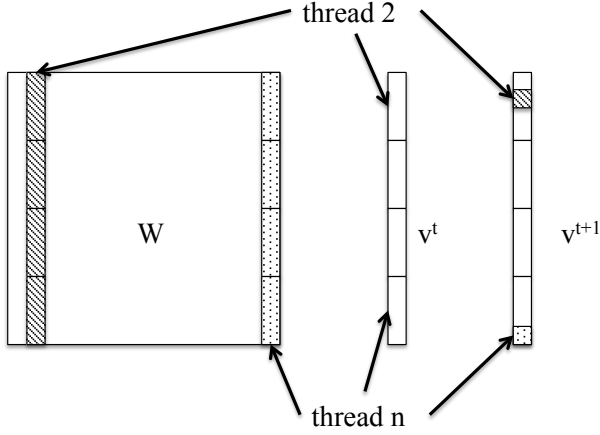
Fig. 8. Illustration of the kernel function for the SUM-OF-MAX rule. A single message retrieval is shown. To update the element v_i^{t+1}, we examine both v^t and the ith column of W.

We do not follow strictly Eq. (6) and Eq. (7) to evaluate a *max* function. Instead, we apply an alternative procedure. Essentially, we check if a neuron receives signals from every cluster; hence, for neuron(i), the ith row of W and v^t require examination. Since W is symmetric and the memory storage on the GPU is column major, we check the ith column of W instead to make the computation more pleasant. To update a neuron value v_i^{t+1}, a dedicated thread i is required, scanning through both v^t and the ith column of W.

Thread i loops through cluster c, from 1 to C. For any positive γ, if neuron(i) belongs to cluster $c = \lfloor \frac{i-1}{L} \rfloor + 1$, we directly set $s_i^t = s_i^t + 1$. Here $\lfloor \cdot \rfloor$ is the standard floor operator. Otherwise, we check within the same cluster, i.e., w_{ji} and v_j^t, where j goes from $(c-1)L + 1$ to cL. The first time we encounter $w_{ji} > 0$ and $v_j^t > 0$, we set $s_i^t = s_i^t + 1$, and proceed to the next cluster $c+1$ without further investigation.

We call this procedure BAIL-OUT-EARLY and favor it over Eq. (6) and Eq. (7) for two reasons:

1) It explicitly clarifies the requirement that every cluster should contribute one and only one signal.
2) It proceeds to subsequent clusters as quickly as possible so that further expensive memory accesses are avoided.

Theorem 1. *The* BAIL-OUT-EARLY *approach is equivalent to* SUM-OF-MAX, *i.e., for any positive γ, given W and v^t,* BAIL-OUT-EARLY *produces the same v^{t+1} as* SUM-OF-MAX.

Proof: For cluster $c = \lfloor \frac{i-1}{L} \rfloor + 1$, there is only one nonnegative $w_{ii} = \gamma > 0$, since by design within the same cluster, a neuron can only receive contributions from itself. The BAIL-OUT-EARLY rule directly sets $s_i^t = s_i^t + 1$, which effectively turns any positive γ into $\gamma = 1$.

For other clusters, w_{ji} is either 0 or 1, depending on whether or not a connection exists between neuron(j) and neuron(i). Notice in either case, v^t is always a binary vector.

Consider $\{a_j = v_j^t w_{ji}\}$, for $j = 1, 2, \cdots, L$. The BAIL-OUT-EARLY approach treats Eq. (6) recursively by implement-

ing the following equation,

$$\max(a_1, a_2, \cdots, a_L) = \begin{cases} a_1 & \text{if } a_1 > 0 \\ \max(a_2, \cdots, a_L) & \text{otherwise} \end{cases},$$

(12)

and treats Eq. (7) by setting $\gamma = 1$. ∎

We deliberately turn any positive γ into 1. The weight matrix W becomes binary valued consequently, which can be more efficiently implemented.

V. PROPERTIES

In this section, we discuss properties of GBNNs and the two retrieval rules introduced in the previous sections. We illustrate these properties via examples and theoretical claims.

A. The SUM-OF-SUM Rule May Oscillate

We first give an example which illustrates that SUM-OF-SUM may oscillate thus not converge. Consider a small network with $C = 3$ clusters, each cluster has $L = 3$ neurons, i.e., 9 neurons in total. We set $\gamma = 1$. There are 4 messages to store: $(1, 1, 1)$, $(2, 2, 1)$, $(3, 2, 1)$ and $(1, 3, 1)$. The test message is $(?, ?, 1)$. Clearly all of the stored messages match the non-erased part of the test message. In such a scenario, we would like that the retrieval rule either returns an *arbitrary* stored message which matches the input, or returns *all* of the stored messages matching the input. Unfortunately, the SUM-OF-SUM rule does not converge to any output. After constructing the network and initializing the neurons to be deactivated for the 1st and 2nd clusters of the test message, we have

$$W = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad v^0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

It is easy to verify that

$$s^0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad v^1 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad s^1 = \begin{pmatrix} 4 \\ 3 \\ 3 \\ 3 \\ 4 \\ 3 \\ 7 \\ 0 \\ 0 \end{pmatrix},$$

$$v^2 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad s^2 = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 3 \\ 0 \\ 0 \end{pmatrix}, \quad v^3 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

The underlined values indicate the maximum value within the same cluster. In this case, $v^3 = v^1$, so that the network does not converge, oscillating between v^2 and v^3 forever.

There is another level of complication: the reinforcement factor γ plays a delicate role in the retrieval procedure. If we increase $\gamma = 2$, then the network converges. However, we will see in Section VII below that enlarging γ leads to a worse retrieval rate in general.

B. The SUM-OF-MAX Rule Converges

We now show that the SUM-OF-MAX (BAIL-OUT-EARLY) rule always converges when all the neurons in erased clusters are initialized to be activated.

Lemma 1. *In the* SUM-OF-MAX *rule, once deactivated, a neuron stays deactivated forever, i.e., if $v_i^t = 0$ then $v_i^{t+1} = 0$.*

Proof: Recall, from Eq. (7), that $v_i^{t+1} = 1$ if and only if $s_i^t = \gamma + C - 1$. Assume in iteration t that neuron(i) is deactivated, i.e., $v_i^t = 0$. Then $v_i^t w_{ii} = 0$. Since the only possible contribution a neuron might obtain from its own cluster is the self loop, $s_i^t = \sum_{c=1}^{C} \max_j(v_j^t w_{ji}) < \gamma + C - 1$, thus $v_i^{t+1} = 0$. ∎

Lemma 2. *A clique is stable, i.e., once all neurons of a clique are activated, they stay activated forever.*

Proof: By definition of a complete sub-graph, all neurons in an activated clique (see Fig. 3) will receive exactly $C - 1$ signals from other clusters and some positive feedback γ. Therefore by Eq. (7), all neurons in the clique stay activated in the next iteration. ∎

Lemma 3. *Given a partially erased message,* SUM-OF-MAX *always produces an ensemble of cliques.*

Proof: As each previously stored message corresponds to a clique, a partially erased message corresponds to parts of the clique, with the neurons in the non-erased clusters activated. SUM-OF-MAX initializes all the neurons in the missing clusters to be activated. Therefore the already activated neurons in non-erased clusters will receive contributions from the missing clusters, staying activated in the next iteration; the neurons in the missing clusters which, together with the already activated neurons in non-erased clusters, form a clique will also receive exactly $\gamma + C - 1$ signals and stay activated in the next iteration. By Lemma 2, the network converges to the ensemble of these cliques. ∎

Theorem 2. *Given a partially erased message, if a neuron is the only one activated in its cluster, it remains activated, i.e., for a given cluster c, if there exists an $i \in \{(c-1)L + 1, \cdots, cL\}$ such that $v_i^t = 1$ and $v_j^t = 0$ for all $j \in \{(c-1)L + 1, \cdots, cL\}$, $j \neq i$, then $v_i^{t+1} = 1$.*

Proof: Suppose, to arrive at a contradiction, that at some point, cluster c has no neuron activated, i.e., $\forall i = (c-1)L + 1, \cdots, cL, v_i^t = 0$, the other clusters will not receive any signal from cluster c. By Eq. (7), every neuron throughout the network will be deactivated in the next iteration. By Lemma 1, the network converges to this all-deactivated state forever, which violates Lemma 3. Therefore, if a neuron is the only one activated in its cluster, it remains activated. ∎

Theorem 3. *For any given probe pattern, the* SUM-OF-MAX *retrieval rule always converges.*

Proof: For a partially erased message, this theorem has already been proved by Lemma 3.

We consider an input probe such that some parts of a previously stored message are modified (corrupted). If the probe can still be explained by a clique in the network, the memory converges to this clique by Lemma 2. If the probe cannot be explained by any clique in the network, the activated neurons in the unchanged clusters cannot receive signals from the corrupted clusters. Hence by Eq. (7), the memory converges to the all-deactivated state. ∎

Since the BAIL-OUT-EARLY rule is equivalent to the SUM-OF-MAX rule (see Theorem 1), we also have the following.

Corollary 1. *For any given probe pattern, the* BAIL-OUT-EARLY *retrieval rule always converges.*

It is worth emphasizing that SUM-OF-MAX converges to an ensemble of cliques by Lemma 3. We can randomly choose one of them as the reconstructed message.

VI. JOINT RETRIEVAL RULE

A. Proposal

We have just seen that SUM-OF-SUM is not guaranteed to converge, whereas SUM-OF-MAX is. In Section VII below we will see that SUM-OF-SUM is generally much faster than SUM-OF-MAX, but the accuracy of SUM-OF-MAX is much better when either the number of stored messages or the number of erased sub-messages increases. It is natural to ask whether we can merge the two rules to obtain the fast reconstruction of SUM-OF-SUM while maintaining the good accuracy of SUM-OF-MAX.

In this section we propose such a hybrid retrieval scheme which combines the best aspects from both procedures. The pseudocode for the joint decoding scheme is given in Algorithm 2. Essentially this decoding algorithm performs one refined iteration of SUM-OF-SUM followed by subsequent, optimized iterations of BAIL-OUT-EARLY until a convergence criterion is satisfied.

B. Justification and Convergence

As mentioned in Section IV, memory access is extremely expensive on GPUs in comparison to on the host CPU. Therefore it is of vital importance that we eliminate any unnecessary memory (read and write) operations. We notice that Lemma 1 and Theorem 2 have crucial implications in designing our new scheme. The former suggests that if $v_i^t = 0$ then there is no need to loop through two long vectors of length n, i.e., v^t and the ith column of W, since we will have $v_i^{t+1} = 0$. Thus, we only need to focus on updating those neuron(i) for which $v_i^t = 1$. In this sense, the currently active neurons can be considered as a candidate pool that needs to be investigated further in the next iteration. The latter suggests that clusters with only one active neuron (including those which are not

Algorithm 2 The joint retrieval scheme.

Input: C – number of clusters

$\quad L$ – number of neurons in each cluster

$\quad e$ – number of erased clusters

$\quad K$ – number of test messages

$\quad W$ – the weight matrix of dimension $n \times n$

$\quad V^0$ – the value matrix of dimension $n \times K$ with each column as a partially erased message for recovery

Output: the recovered matrix V^t

1: initialize all the neurons inactive in erased clusters
2: $S^0 = WV^0$ {SUM-OF-SUM}
3: **for** each thread **do** {eK threads in parallel, each for an erased cluster}
4: $\quad$ check S^0 and keep neurons with $C - e$ signals activated
5: **end for**{V^1 obtained}
6: sparsify W to obtain W' {use W' afterwards}
7: t = 0
8: **repeat** {BAIL-OUT-EARLY (SUM-OF-MAX)}
9: $\quad$ t = t+1
10: $\quad$ **for** each thread **do** {eLK threads in parallel, each for a neuron in erased clusters of different messages}
11: $\quad\quad$ keep off deactivated neurons, otherwise apply the BAIL-OUT-EARLY scheme
12: $\quad$ **end for** {V^{t+1} obtained}
13: **until** $V^{t+1} == V^t$ {only check erased clusters}

---

erased in the test message) will not change during decoding. Hence, we only need to update neurons in erased clusters that have not yet reached convergence. In general, this notion of "freezing good clusters" can also be justified as preventing good neurons from being altered undesirably by any retrieval procedure.

One final but important consideration is the all-activated initialization scheme. Although it is crucial for the correctness of the SUM-OF-MAX rule, it also introduces too many candidates from the beginning. We will show a motivating example later in Section VII-E. Fortunately, SUM-OF-SUM can help us bypass this particular problem.

Theorem 4. *The first iteration of* SUM-OF-SUM *affects neither the correctness of* SUM-OF-MAX *nor its convergence.*

Proof: For correctness, let us revisit the SUM-OF-SUM rule. The only problem which makes SUM-OF-SUM inferior to SUM-OF-MAX is that during the retrieval procedure, as in Fig. 4, it is possible for multiple neurons to be activated simultaneously in one cluster *without* regulation, which in turn propagates the error or even causes oscillation. However, during the initialization phase, if we deactivate all the neurons in erased clusters, preserving good clusters only, by definition there will be at most one activated neuron per cluster. The aforementioned flaw does not exist anymore.

For convergence, recall a clique structure in Fig. 3. For a given message with e clusters erased, there are $C - e$ good neurons sending out signals, therefore the desired neuron in the erased clusters should receive $C - e$ signals exactly. After

one iteration of SUM-OF-SUM, we only keep these neurons with $C - e$ signals in the candidate pool. The sole effect of the first iteration SUM-OF-SUM is to shrink the pool size, with the convergence untouched. ∎

Ideally, there should be only one such neuron per erased cluster in the candidate pool, rather than L candidates for SUM-OF-MAX, with two exceptional cases.

- There are two memorized messages m_1 and m_2 which only differ in erased clusters, e.g., we have $m_1 = (1, 3, 1)$ and $m_2 = (1, 3, 2)$, and the test message is $m = (1, 3, ?)$. In this case, both neuron 1 and 2 in the erased cluster will be present in the pool.
- Artificial clique structures. While storing messages, we print a clique structure onto the network for each message stored. At some point, we may introduce artificial cliques; i.e., we will create a clique in the network corresponding to a message which was never stored, where different edges in this clique were added for different stored messages. In another word, a previously stored message corresponds to a clique, but not vice versa.

We argue that for a relatively large network and a reasonable number of messages to memorize, the candidate pool size is sufficiently small.

Corollary 2. *The new joint retrieval scheme always converges.*

Proof: The joint scheme invokes one iteration of SUM-OF-SUM followed by iterations of BAIL-OUT-EARLY. According to Theorem 4, SUM-OF-SUM effectively reduces the size of the candidate pool, with the network's convergence property untouched. The joint scheme thus always converges by Theorem 3. ∎

Combining all these factors, we propose the joint scheme as in Algorithm 2.

VII. EXPERIMENTS

In this section, we compare SUM-OF-SUM and SUM-OF-MAX using the different acceleration approaches discussed previously in Section IV and Section VI. We show that a significant performance gain is achieved in terms of running time after parallelizing the retrieval procedure and applying our new joint retrieval scheme.

All the CPU experiments are executed on a 2.6GHz AMD Phenom (tm) 9950 Quad-Core Processor with 16GB of RAM, and all of the GPU experiments are executed on an NVIDIA C1060 card, which runs at a frequency of 1.3GHz with 4GB memory and has 30 stream multiprocessors. In order to make as far a comparison as possible, our CPU code makes use of the fastest linear algebra package available – Armadillo library [21], linked to BLAS and LAPACK, for optimized linear algebra operations.

A. SUM-OF-SUM *versus* SUM-OF-MAX

First, we compare SUM-OF-SUM and SUM-OF-MAX. In this experiment, we have $C = 8$ clusters with $L = 128$ neurons each, and reinforcement factor $\gamma = 2$. We randomly generate and store 5000 messages, each of which consists of 8 sub-messages uniformly sampled from the integers 1 to

128. After the storing stage, we randomly select 3000 out of the 5000 stored messages, erase some parts of them and try to retrieve the erased messages from the GBNN associative memory. We refer to this experiment setting as Scenario 1. Since SUM-OF-SUM does not necessarily converge, we set the maximum number of iterations to 20. We vary the number of erased clusters, and plot the retrieval rate, i.e., the fraction of successfully retrieved messages, in Fig. 9.
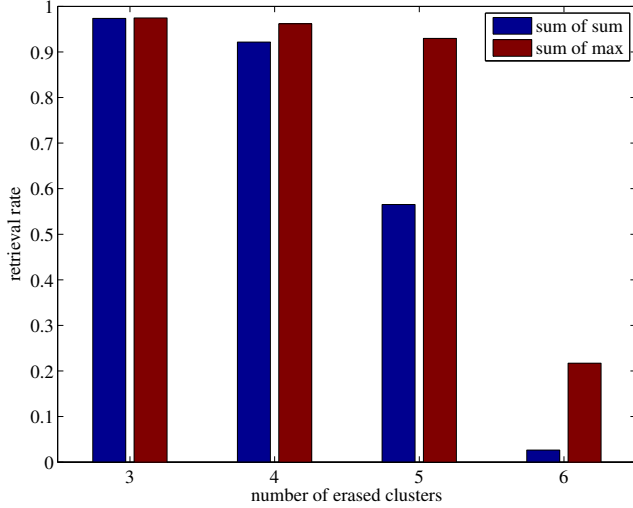

Fig. 9. Comparison between the SUM-OF-SUM and SUM-OF-MAX retrieval rules. We have $C = 8$, $L = 128$, 5000 stored messages and 3000 test messages. We vary the number of erased clusters, and plot the retrieval rate.

Observe that when the number of erased clusters is relatively small (3 erased clusters), both rules perform equally well above 97%. As the number of erased clusters increases, although both rules make more errors, the performance of SUM-OF-SUM degrades more drastically than that of the SUM-OF-MAX rule. When 5 out of 8 clusters are erased, SUM-OF-SUM can only recover slightly above 50% of the messages, while SUM-OF-MAX still recovers over 90%. If 6 clusters are erased, SUM-OF-MAX is still able to retrieve over 20%, which is significantly higher than SUM-OF-SUM.

B. Influence of γ

Second, we explore the subtle dependence of the retrieval rate on the value of the reinforcement factor γ used in the SUM-OF-SUM rule. We plot the trend for different γ in Fig. 10 using the same experiment Scenario 1 as above. In general, increasing γ hurts the retrieval rate with the only exception of $\gamma = 0$, which suggests that $\gamma = 1$ can be used as a default value.

C. CPU versus GPU

Next, we consider the improvements in runtime achieved by running both rules on a GPU versus on a CPU. A larger network is simulated in this case. We have $C = 16$ clusters with $L = 512$ neurons each, out of which 7 clusters are erased. We generate and store 50000 random messages, and we use a random subset of 30000 of these to test. We refer to this
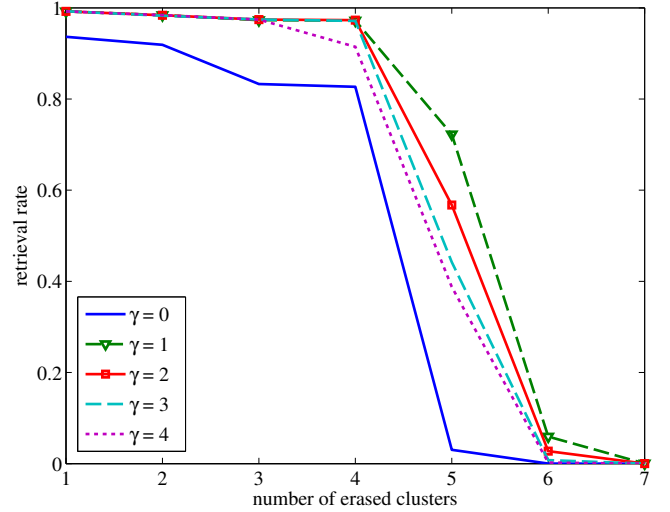

Fig. 10. Subtle dependence between the retrieval rate and the reinforcement factor γ. We have $C = 8$, $L = 128$, 5000 stored messages and 3000 test messages. We vary the number of erased clusters, and plot the retrieval rate.

experiment setting as Scenario 2. The runtime, in seconds, of both parallel decoding (i.e., decoding a batch of messages concurrently) and serial decoding (i.e., decoding message one after another) on both GPU and CPU are shown in Fig. 11.
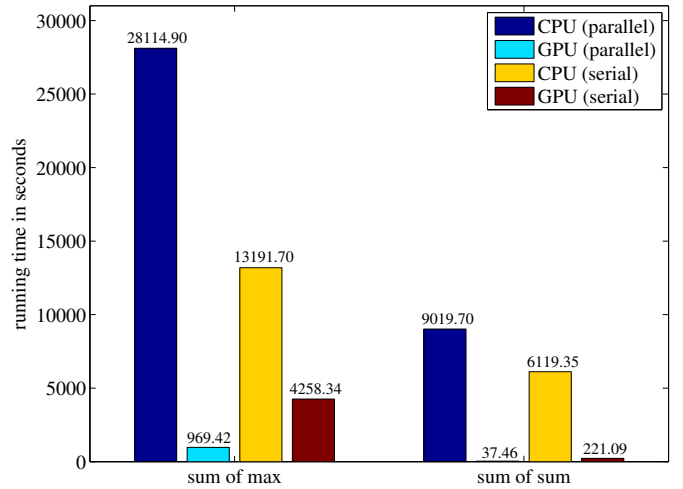

Fig. 11. Running time in seconds of both rules running on CPU and GPU respectively. We have $C = 16$, $L = 512$, 50000 stored messages and 30000 test messages.

We make three observations:

1) For each CPU versus GPU and parallel versus serial decoding configuration, the SUM-OF-MAX rule is always significantly slower than SUM-OF-SUM. For now, let us keep in mind that the fastest retrieval configuration of this entire experiment is roughly 40 seconds for SUM-OF-SUM parallel decoding on a GPU. We have previously seen that SUM-OF-MAX leads to a much better retrieval accuracy, and so below we focus on achieving the accuracy of the SUM-OF-MAX method while improving its runtime.

2) In each group, the bars at the 1st and 3rd locations are results for the CPU implementation, and the 2nd and 4th bars show results for the GPU implementation. Comparing each adjacent pair, we see that the GPU versions consistently run much faster than CPU, as expected. The GPU accelerations without any further optimization are respectively (from left-to-right) $29\times$, $3\times$, $240\times$ and $28\times$ faster.

3) Parallel decoding is faster than serial decoding on GPU, while the situation reverses on CPU. This is reasonable, since parallel decoding can take full advantage of the GPU's computing power. However, in the CPU case, if we consider a bundle of K messages, even if only one message does not converge, all K messages will be updated. On the other hand, with serial decoding, the retrieval rule will stop as soon as each individual message converges.

D. Further Accelerating the SUM-OF-MAX Rule

In Fig. 12 we show the effect of applying the different techniques discussed in Sections IV and V to accelerate the SUM-OF-MAX rule on a GPU. Although all of the techniques combined reduce the runtime eightfold, from roughly 4000 seconds to 500 seconds, the SUM-OF-MAX rule still cannot compete with the 40-second spec of the SUM-OF-SUM rule, which is highlighted in yellow and bold font in the figure. However, the proposed joint scheme cuts the record by another half, achieving the fastest runtime of only 17.38 seconds for Scenario 2.

In Fig. 11, the faster configuration for SUM-OF-MAX on CPU is the serial decoding scheme, to which we compare, our joint scheme achieves a $760\times$ speedup while retaining the decoding accuracy.
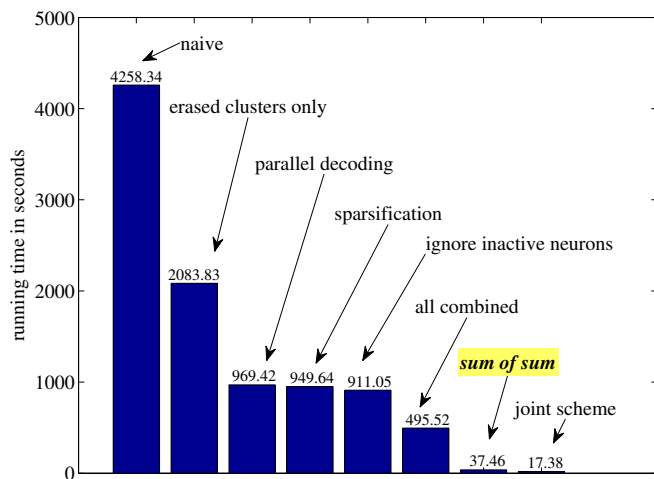

Fig. 12. Running time in seconds for different acceleration tricks. We have $C = 16$, $L = 512$, 50000 stored messages and 30000 test messages. The first six bars are for SUM-OF-MAX. We plot the performances for different tricks and all combined. We also plot the record of SUM-OF-SUM and our joint scheme at the end for comparison.

E. Motivation for Combining Rules

Here we provide an example to help better illustrate why the joint scheme achieves a significant speedup. We again use Scenario 2 and apply all of the acceleration techniques discussed in Section IV. We initialize the matrix V^0 according to SUM-OF-MAX, so that all neurons in clusters corresponding to erased sub-messages are activated, and only one neuron within each cluster corresponding to a non-erased sub-message is active. Fig. 13 depicts a typical run of the experiment. Fig. 13a shows the total runtime spent in each iteration of the SUM-OF-MAX decoding. Clearly the majority of the runtime is spent in the 1st iteration; this occurs because there are too many unnecessary active neurons in erased clusters, and the SUM-OF-MAX rule demands time to process each one of them. Fig. 13b shows the number of test messages (out of 30000) which have converged after each iteration.

F. New Joint Scheme

Finally, we demonstrate the behavior of the joint decoding scheme across a range of experimental settings. Fig. 14 shows the runtime (in seconds) and retrieval rate compared with SUM-OF-SUM and SUM-OF-MAX for both of Scenarios 1 and 2, while varying the number of erased sub-messages. The spikes in runtime for SUM-OF-MAX and for the joint scheme in Fig. 14a are due to the fact that decoding becomes more difficult as the number of erased clusters increases, consequently more iterations are required in these cases. In these settings (6 out of 8 clusters erased for Scenario 1, and 14 out of 16 clusters erased for Scenario 2), although the SUM-OF-SUM rule is only a bit faster than SUM-OF-MAX and the joint scheme, the retrieval rate is significantly lower. Another reason that SUM-OF-SUM runs faster here is due to the limit on the number of iterations which we impose in our experiments. Note that increasing this limit does not improve the retrieval rate, but it can make the runtime arbitrarily worse because SUM-OF-SUM oscillates. Also observe that in both Fig. 14b and Fig. 14d, the retrieval rates of SUM-OF-MAX and the joint scheme are identical. In Fig. 14d, all three approaches achieve effectively a 100% retrieval rate for up to 13 erased clusters. This is because the number of messages stored (50000) is relatively small for this network. If this number increases, the deviation in retrieval rate between the joint scheme (as well as SUM-OF-MAX) and SUM-OF-SUM will be more pronounced. We conclude from Fig. 14 that the joint retrieval scheme combines the benefits of both existing rules, achieving fast decoding while also maintaining a high retrieval rate.

VIII. SUMMARY

In this work, we present optimized implementations of the Gripon-Berrou neural network associative memory on a GPU. We analyze two existing retrieval rules, namely SUM-OF-SUM and SUM-OF-MAX. We show that SUM-OF-SUM may lead to network oscillation, however, we manage to prove the convergence of SUM-OF-MAX. In order to achieve the full speedup, we combine the two retrieval rules and propose a hybrid retrieval scheme, minimizing the unnecessary computation burdens. The experimental results show an exciting

(a) running time in seconds for each iteration

(b) accumulate number of convergent messages after each iteration
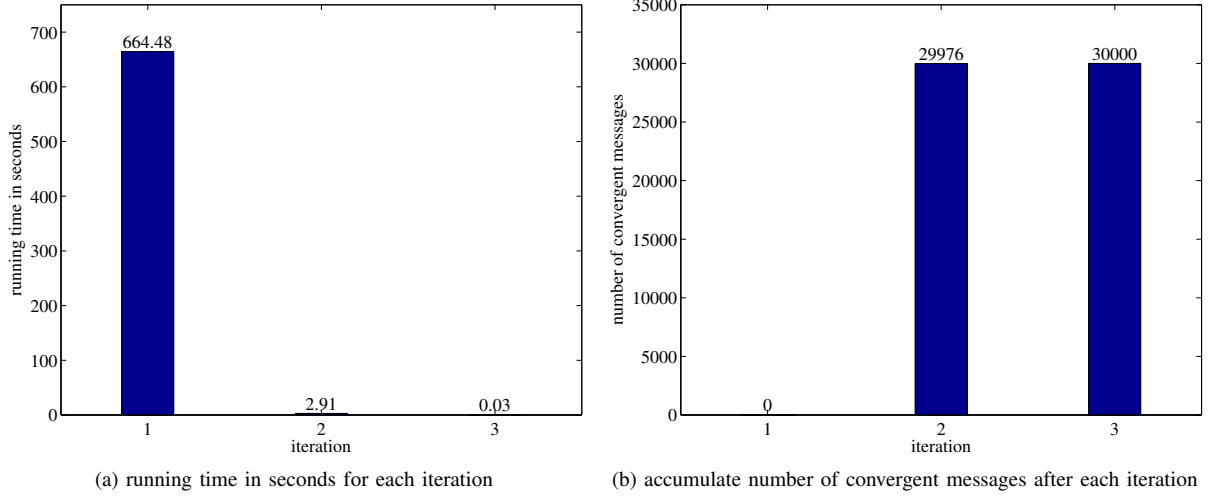
Fig. 13. A typical run of the vanilla SUM-OF-MAX with all the neurons in the erased clusters activated. We experiment Scenario 2 where we have $C = 16$ clusters, with $L = 512$ neurons each, 50000 messages to memorize and 30000 messages to test. (a) shows the running time in seconds for each iteration. (b) shows the accumulate number of the messages that converges after each iteration.

(a) running time for Scenario 1

(b) retrieval rate for Scenario 1

(c) running time for Scenario 2
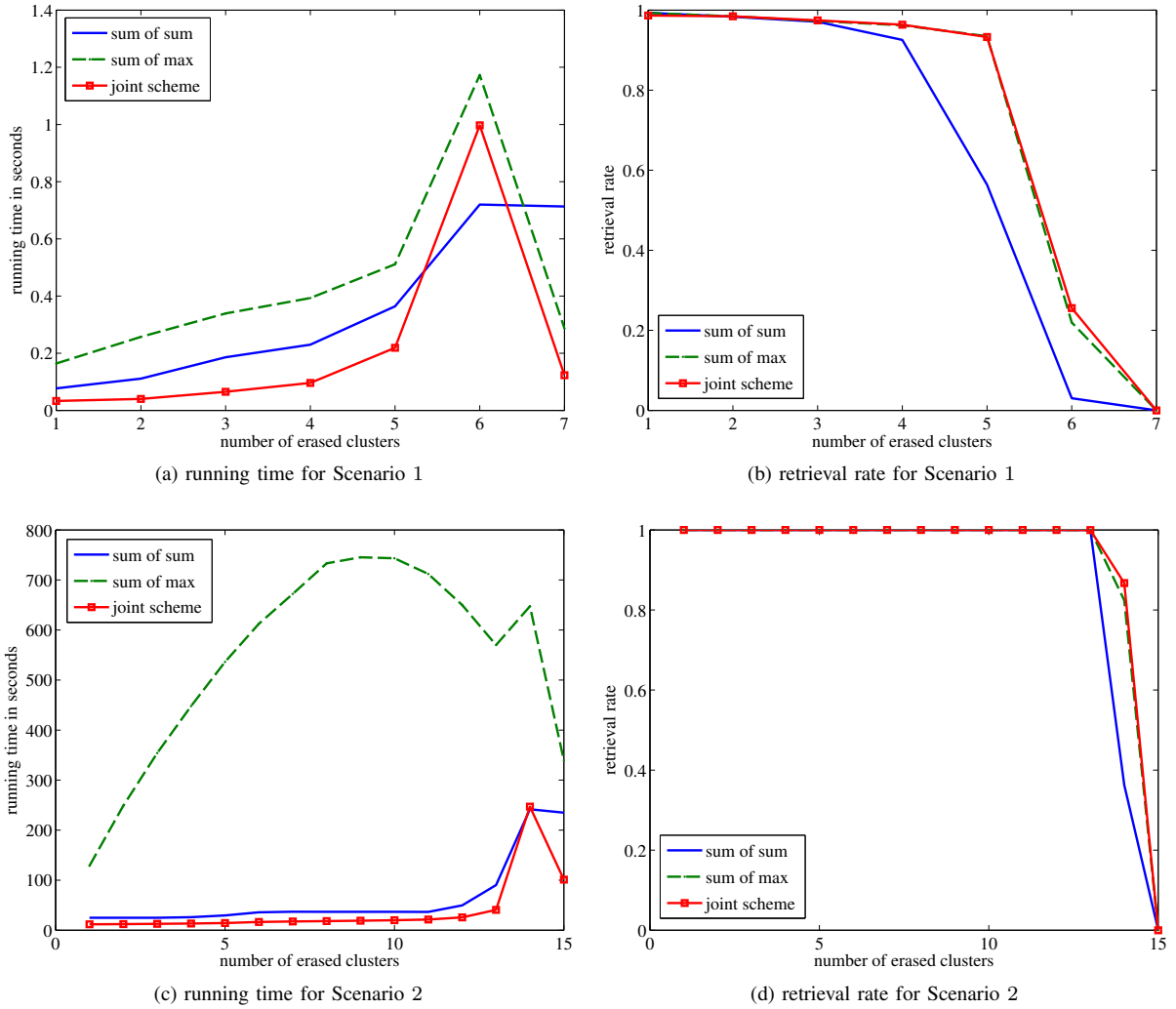
(d) retrieval rate for Scenario 2

Fig. 14. The behavior of the joint retrieval scheme in general. Both running time in seconds and retrieval rate are plotted respectively as the number of erased clusters increases. We set $\gamma = 2$ and the maximum number of iterations allowed is 20. (a) and (b) refer to Scenario 1 where there are 8 clusters with 128 neurons each, 5000 messages to memorize, and 3000 messages to test. (c) and (d) refer to Scenario 2 where there are 16 clusters with 512 neurons each, 50000 messages to memorize, and 30000 messages to test.

acceleration against a CPU implementation using an optimized linear algebra library.

GBNNs embrace a LDPC-like sparse encoding setup, which makes the network extremely resilient to noises and errors. As associative memories serve as building blocks for many machine learning algorithms, we hope the parallel scheme proposed here can help in paving the path to more widespread adoptions of large scale associative memory applications.

In the future, we will try to find other retrieval schemes. For instance, since SUM-OF-SUM runs orders of magnitude faster, emulating SUM-OF-MAX using SUM-OF-SUM seems to be another sensible choice, and our initial results in this direction are promising. We may also seek the way to generalize GBNN and extend the sparse neural network's use in tasks other than associative memory, e.g., classification and regression.

REFERENCES

[1] S. Rajasekaran and G. Pai, *Neural Networks, Fuzzy Logic and Genetic Algorithms: Synthesis and Applications*. PHI Learning Pvt. Ltd., 2004.

[2] F. Rosenblatt, *Principles of Neurodynamics*. Spartan Book, 1962.

[3] T. Kohonen, *Self-Organization and Associative Memory*. Springer-Verlag, 1988.

[4] C. S. Lin, D. C. P. Smith, and J. M. Smith, "The design of a rotating associative memory for relational database applications," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 53–65, march 1976.

[5] L. Bu and J. Chandy, "FPGA based network intrusion detection using content addressable memories," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, april 2004, pp. 316–317.

[6] K.-J. Lin and C.-W. Wu, "A low-power CAM design for LZ data compression," *IEEE Transactions on Computers*, vol. 49, no. 10, pp. 1139–1145, october 2000.

[7] H. Zhang, B. Zhang, W. Huang, and Q. Tian, "Gabor wavelet associative memory for face recognition," *IEEE Transactions on Neural Networks*, vol. 16, no. 1, pp. 275–278, 2005.

[8] J. A. Anderson and J. Davis, *An Introduction to Neural Networks*. MIT Press, 1995, vol. 1.

[9] A. Jain, J. Mao, and K. Mohiuddin, "Artificial neural networks: A tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, march 1996.

[10] J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[11] ——, "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences*, vol. 81, no. 10, pp. 3088–3092, 1984.

[12] B. Kosko, "Bidirectional associative memories," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 18, no. 1, pp. 49–60, 1988.

[13] V. Gripon and C. Berrou, "A simple and efficient way to store many messages using neural cliques," in *IEEE Symposium on Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB)*. Paris, France: IEEE, 2011, pp. 1–5.

[14] ——, "Sparse neural networks with large learning diversity," *IEEE Transactions on Neural Networks*, vol. 22, no. 7, pp. 1087–1096, 2011.

[15] C. Berrou and V. Gripon, "Coded hopfield networks," in *International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, september 2010, pp. 1–5.

[16] V. Gripon and C. Berrou, "Nearly-optimal associative memories based on distributed constant weight codes," in *Information Theory and Applications Workshop (ITA)*. San Diego, CA, USA: IEEE, 2012, pp. 269–273.

[17] X. Jiang, V. Gripon, and C. Berrou, "Learning long sequences in binary neural networks," in *International Conference on Advanced Cognitive Technologies and Applications*, 2012, pp. 165–170.

[18] B. K. Aliabadi, C. Berrou, V. Gripon, and X. Jiang, "Learning sparse messages in networks of neural cliques," ACM Computing Research Repository, 2012, `arXiv:1208.4009v1 [cs.NE]`. [Online]. Available: http://arxiv.org/abs/1208.4009v1

[19] H. Jarollahi, N. Onizawa, V. Gripon, and W. Gross, "Architecture and implementation of an associative memory using sparse clustered networks," in *IEEE International Symposium on Circuits and Systems (ISCAS)*. Seoul, Korea: IEEE, 2012, pp. 2901–2904.

[20] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 948–960, 1972.

[21] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," Technical report, NICTA, Tech. Rep., 2010.